Constrained Decision Revision

Charles Petrie
MCC AI Lab
3500 West Balcones Center Drive
Austin, TX 78759
petrie@mcc.com

Abstract

This paper synthesizes general constraint satisfaction and classical AI planning into a theory of incremental change that accounts for multiple objectives and contingencies. The hypothesis is that this is a new and useful paradigm for problem solving and re-solving. A truth maintenance-based architecture derived from the theory is useful for contingent assignment problems such as logistics planning.

Introduction

There is a large class of practical planning problems that require not only general constraint satisfaction but also search by reduction and incremental replanning lacking in classical constraint satisfaction formalisms. Classical AI planning, in the form of hierarchical nonlinear planning, provides a useful method of search reduction, but an inadequate representation for consistency and replanning.

Such problems, which we call *Constrained Decision Problems* (CDPs), are characterized by multiple objectives, a large space of acceptable solutions from which only one is sought, general constraints, succinct representation by decomposable goals, and replanning: the necessity to react precisely to changed conditions. Travel planning is a prototypical example of a CDP: a plan consists of assignments, rather than actions, satisfying constraints and depending upon assumptions that may change during planning and execution.

As a simple example, suppose we want to represent the following travel planning knowledge. Given the goal of traveling from one place to another, there are three methods of doing so: flying, and taking a bus or a taxi. Flying is preferable if the trip is more than 100 miles. Otherwise a taxi is preferable. If we choose to fly, we decompose the problem into two subproblems: choosing a flight to the nearest airport and finding a way from the airport to our final destination. We prefer the cheapest flight. There are contingencies. Individual flights and buses can be canceled. Airports can be closed. Taxis and cars may be unavailable. All travel methods have costs. There is a travel budget constraint.

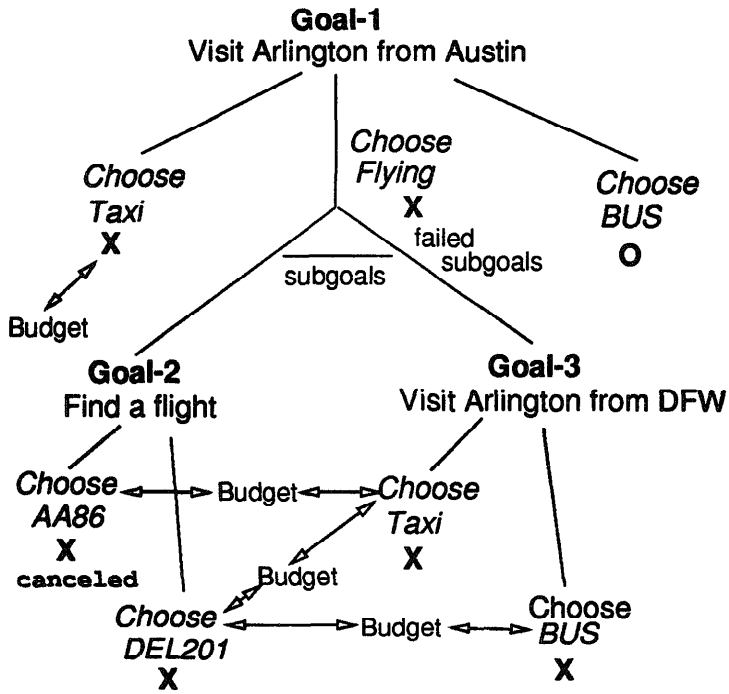

Figure 1: Problem State

The following problem solving scenario indicates the desired use of this knowledge by a planner. The goal was to go from Austin to Arlington, Texas. The planner made a decision to fly. This resulted in two subgoals: finding a flight to the DFW airport, and getting from the airport to the suburb of Arlington. Initially, based on cost, flight AA86 and the airport shuttle bus were chosen, the taxi being preferable but too expensive. Then flight AA86 was canceled. The planner noted that part of the plan no longer worked and attempted to fix just that part.

Another flight, DEL201, was tried but it was too expensive, even using an airport shuttle bus. The de-

cision to fly had to be retracted, since there were just the two flights. A taxi was too expensive to use for the whole trip, so the plan was to go by bus from Austin to Arlington. Figure 1 shows this state of problem solving. **X** denotes an alternative that was tried and either invalidated or rejected due to the budget constraint. **O** denotes the current solution.

Now, if flight AA86 is rescheduled in this state, the planner should notify the user of this opportunity and allow the user to drop back to this preferable state if we desired. Similarly, the user would like to know by how much the budget should be increased to afford flight DEL201. And the planner should inform the user if that option is possible upon discovery of a new shuttle bus price. In no case do we want our current plan changed automatically. This might be only part of a larger plan the user doesn't want rearranged.

No previous formalism is suitable for providing the computer support illustrated above. Alternatively, *ad hoc* applications with such functionality quickly become too difficult to manage for more complex problems. Existing shells do not supply the right primitives to reduce the complexity. In this paper, we provide a formalism and an architecture that simplifies representing and solving such CDPs. The example above is represented in the architecture by declaring two types of goals, four operators for satisfying them, one constraint, and three preference rules. The contingencies are represented by operator conditions.

Why Not Constraint Satisfaction?

CDPs are typically constrained task/resource assignment problems that occur over time, such as construction planning, teacher/course assignment, and logistics. However, constraint satisfaction or integer programming techniques are unsuitable for these problems.[1]

First, global objective functions of integer programming are inadequate for multiple objectives. Second, replanning from scratch is inadequate because of the desirability of minimal change to a plan and the lack of control integer programming provides.

Third, to formulate a constraint satisfaction or integer programming problem, one must enumerate all of the possibilities/variables. However, CDPs are constructive in that new objects are typically created at runtime; e.g., a subcomponent of a design or a leg of a trip. But an attribute together with an object defines a variable to which a value may be assigned; e.g, the cost of a particular trip leg. Thus defining all variables means anticipating the creation of all possible objects, whether used in the final plan or not. This is a difficult task in the formulation of constructive problems. Yet the integer programming and constraint satisfaction formalisms needlessly require this work.

[1]Integer programming is a special case of constraint satisfaction with an additional objective function to be maximized.

Generalizing Classical Planning

CDPs only require a single solution (or a few) out of a large space of acceptable ones. Hierarchical, nonlinear planning (HNP) is a good problem solving technique for CDPs. In *hierarchical* planning [10], a plan can progress from higher to lower levels of abstraction through plan transformations. In *nonlinear* planning [11], actions need not be totally ordered with respect to time, and ordering need be introduced only as necessary [12]. If HNP is used in heuristic-guided depth-first search, problem decomposition is provided, only a few solutions are explored, and objects and assignments are created only when the required level of specificity is reached. But classical HNP does not provide for the general constraint satisfaction needed by CDPs, where constraints on assignments are frequently boolean or global resource usage.

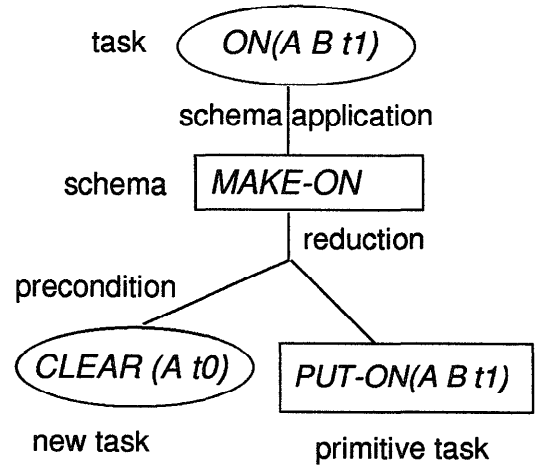

Figure 2: HN Planning

Classical HNP defines the application of *schemas* to *tasks*[4]. We generalize this as the application of operators to goals, perhaps creating new subtasks and/or actions. We generalize this to say that the application of an operator to a goal may result in subgoals and/or variable assignments. Figures 2 and 3 illustrate two major differences between HNP schema reduction and CDR decision making. The first is that the choice of an operator to reduce a goal from a conflict set of applicable operators is a *decision* with a rationale. Second, actions and orderings are represented as general variable assignments. More typically in a CDP, an assignment is something like the cost of the leg of a trip or, in another example, the assignment of a professor to teach a course. The importance of representing a decision rationale and assignments is described below.

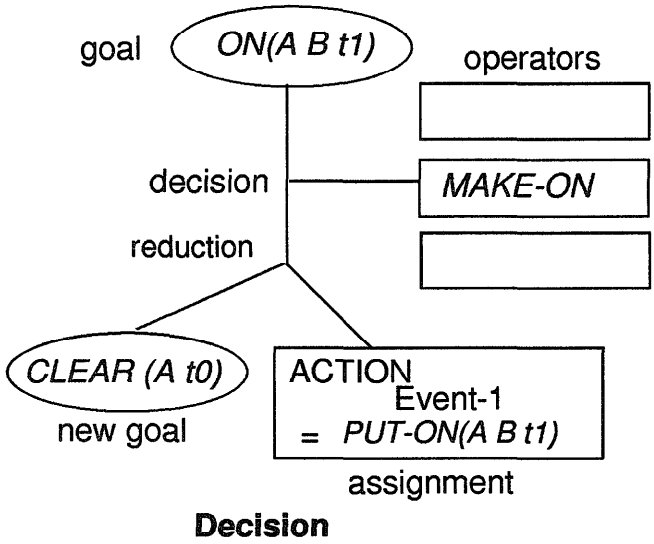

Figure 3: CDP Solving

Decision Revision

HNP proceeds by reducing tasks until nothing but primitive tasks are left. CDP solving proceeds similarly. If an operator has been validly applied to a goal, then that goal is *reduced* and need not be further achieved. When no goal is left un-reduced, all goals are satisfied and a complete solution consisting of assignments is achieved. Constraints are consistency conditions, initially satisfied, that may be violated by decisions. Goals are patterns to be reduced by operators. Consistency is directly evaluated by the planner whereas goal satisfaction is assumed if an operator has been applied. We can then cast the design of a plan as a series of such operator applications, i.e., decisions. Backtracking and replanning are cases of decision revision.

Backtracking

In classical HNP, schemas have pre- and post-conditions that might conflict with each other. HNP is specialized to satisfy the implicit binary constraints defined by these conditions. This is optimal for temporal action sequence consistency, but not for general constraints. E.g., gadgets may not be configured with widgets, no one may teach more than two classes, and the budget must not be exceeded. In contrast to STRIPS-style conditions, in CDPs, we require that all possible conflicts among decisions be explicitly stated as constraints among resulting assignments. (*Constraints* are encoded as Horn clauses and indexed to the variables in their domain in the implementation.) This is no more restrictive than the STRIPS requirement to state all relevant conditions, and it allows more general constraints to be represented.

Using the general notion of variable assignment allows the use of general techniques of constraint satisfaction, including variable choice, delayed resolution, propagation, and dependency-directed backtracking. When some set of assignments is determined to conflict with a constraint, it can always be resolved by retracting some minimal set of decisions.[2] That is, decisions are a distinguished set of *assumptions* that can be retracted at will. This retraction of decisions is backtracking. Because only decisions are considered that support the assignments in the conflict, it is dependency-directed backtracking.

Replanning - Catastrophes

An expectation about the environment in which plan actions will be taken is another kind of *assumption*. Associated with each operator is a set of *admissibility* conditions. A decision is valid only as long as these hold. The admissibility of one decision may not depend upon the validity of others - that would be a constraint. But admissibility may depend upon assumptions about the plan execution environment, such as the availability of a resource. Events may prove such expectations wrong - a contingency occurs. E.g., a flight is canceled or a professor falls ill. In such a case, some decisions will become invalid. This may cause the loss of satisfaction of some goal - a catastrophe.

It is important to represent these assumptions differently from the contingent nature of decisions. The latter can be retracted at will (until committed) in the face of constraints. But to change our assumptions about the environment for the sake of planning convenience is to indulge in the wishful thinking noted by Morris [5]. Backtracking must be restricted to decision retraction: environmental assumptions can only be changed because of new information.

Replanning - Opportunities

There may be a partial ordering associated with each set of operators, and operator instances, applicable to a goal. Based on heuristics, this ordering provides a rationale for the decision to apply one, from the conflict set of possibilities, to the goal. (These *preferences* are also encoded as Horn clauses in the implementation.) If the goal is viewed as an objective, this partial ordering defines the local *optimality* of different ways of satisfying the objective. This optimality may be conditional on environmental expectations in the same way that admissibility was. If they change, the ordering may change. Possibly, some decision ought to be revised. For instance, suppose that airline flights change costs and another is cheaper than the one currently planned. Or a previously preferred flight that was canceled is rescheduled.

Optimality also depends upon goal validity, which in turn may also depend upon events. If enrollment is

[2]This is easily shown in the formal CDP theory given in [9].

less than expected, a goal of providing six logic courses may no longer be valid. The decision to provide a sixth course may no longer be optimal, though it is still valid. If a friend will be away on vacation while we are traveling, we do not need to include visiting his city in our plans.

Backtracking and Replanning

Catastrophes and opportunities can also result from backtracking. Whether a decision is invalidated by an external event or rejected as a part of constraint satisfaction, the invalidation of a dependent assignment may mean that one or more goals is no longer reduced or satisfied - a catastrophe. The invalidation of a subgoal because of the rejection of the parent decision may mean the loss of optimality for subsequent decisions, which are opportunities to improve the plan, perhaps by simple elimination of some assignments.

An important subcase of an opportunity caused by backtracking is when the best decision, say Λ_a, for goal a is rejected because it conflicts with some second decision Γ_b for some other goal b. Then the next decision made, say Υ_a, is optimal only with respect to the rejection of Λ_a. But if Γ_b, originally in conflict with Λ_a, is itself later rejected, then the current decision, Υ_a, has lost its rationale for optimality and represents an opportunity.

The important point is that backtracking and replanning require the same change management services. The change must be propagated to the dependent assignments and subgoals, and the catastrophes and opportunities detected. Thus both backtracking and replanning are special cases of general decision revision.

Incremental Revision

CDPs requires replanning to be done incrementally. First, replanning and planning must be interleaved for problems in which execution begins before the plan is complete. This alone requires that replanning proceed in the same incremental fashion as does depth-first, heuristic guided search that is appropriate for CDPs.

It is also the case that the notion of replanning from scratch does not capture the constraints over the planning work itself. It is easy to construct integer programming task/resource assignment examples in which the unexpected unavailability of a resource would cause all assignments to be changed in order to satisfy the global objective function. But this ignores planning work. Typically, there are constraints over how much change is acceptable in a plan. A department chairman will not want to rearrange every professor's schedule if it is not necessary to do so. A less than optimal solution that adequately responds to the catastrophe may be the best solution if it minimally perturbs the plan.

This consideration is especially important for multiple objectives with local optimalities. While it is important to know about opportunities to improve the satisfaction of some objective without making that of another worse, it is not always best to revise the plan to take advantage of them. A complex design should not be massively changed because of a small improvement made possible by the use of a newly available component.

Computational Support

The CDP model describes how a system should react to change. Some response is syntactically determined by the nature of the dependencies. Other change requires semantic reasoning: the syntactic task is only to provide information for the reasoning and respond to the conclusion. The reasoning may be performed by the user or heuristic rules. Backtracking and replanning differ in the source of the change and the initial response.

In CDP replanning, contingencies cause invalidation of admissibility conditions for decisions, making the decisions invalid. Abstractly, a contingency is a hard, unary constraint: there is no choice about how to respond. If the flight is canceled, the system can syntactically determine that the decision to use that flight must be invalidated. The computational support is indexing of decisions by admissibility conditions.

Rejection in backtracking to resolve constraint violations also invalidates decisions. General constraint satisfaction generally requires reasoning about which decisions to retract. If we can't afford to go to both cities, we must decide which to cut. So in backtracking, the initial syntactic task is only to present the relevant alternatives. Retraction is controlled by the user or rules. The primary computational support is indexing assignments by their supporting decisions and relevant goals.

Once a decision has been invalidated, whether because of backtracking or replanning, the syntactic task is the same: propagation of the invalidation and detection of catastrophes and opportunities. The computational support for decision invalidation in both cases is indexing decision, decision rationale, and goal validity by supporting decisions.

In particular, for catastrophes, the computational support required is to identify the assignments and the transitive closure of goals no longer reduced or satisfied because of the invalidity of affected decisions. Associated assignments and goals must lose their validity. For opportunities, the task is to identify which decision rationales, or goal validities, are invalidated. Although this information should be conveyed to the user, the plan should not be automatically revised, as discussed in the previous section.

Detecting opportunities is the particularly difficult and important search task of context maintenance. Surprisingly, it turns out to be equivalent to tracking the *pareto optimality*[3] of a solution for multiple objectives. No part of a pareto optimal solution can be improved without making some other part worse.

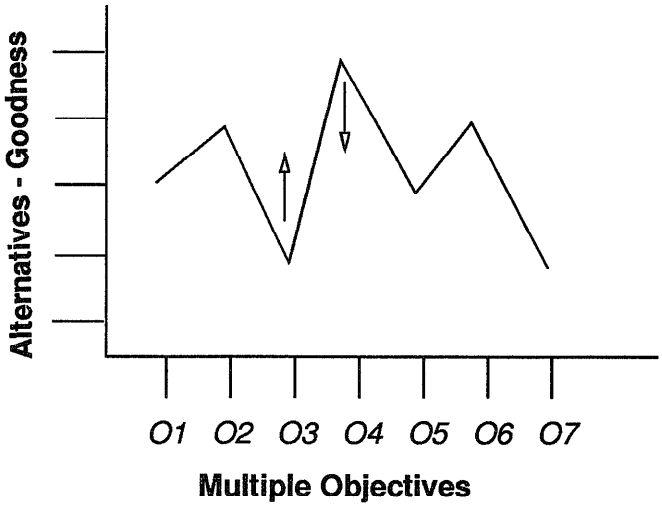

Figure 4: Pareto Optimality

As illustrated in Figure 4, if the satisfaction of objective $O3$ can only be obtained at the expense of the satisfaction of some other, say $O4$, and a similar situation exists for all other parts of the solution, then it is pareto optimal. On the other hand, if we could improve the satisfaction of $O3$ while not affecting that of any other objective, then the solution is not pareto optimal and we have detected an opportunity.

This special case, discussed above, of an opportunity caused by backtracking turns out to correspond to and provide semantics for a known computational technique: generating contradiction resolution justifications in a truth maintenance system (TMS)[6, 8].

Dependencies

In [9], we give a formal theory of CDP solving. The entailments in that theory make precise the ontology and dependencies discussed above. The theory provides semantics for the dependencies in a computational architecture, REDUX, for revising plan states. REDUX uses a TMS to represent the dependencies that correspond to the entailments that represent the CDP model. Rather than use entailments, we use the TMS notation to describe partially the CDP dependencies. These dependencies are also the implementation encoding.

In passing, we note that truth maintenance is a natural candidate for a mechanism to propagate the effects of change in a design or plan, given some reasonable tradeoffs between completeness and tractability. But truth maintenance has not been used extensively for design and planning, despite many proposals for doing so. We claim that the problem is a lack of semantics for dependencies and that the CDP model provides these [7].

Figures 5 and 6 show templates for most of the de-

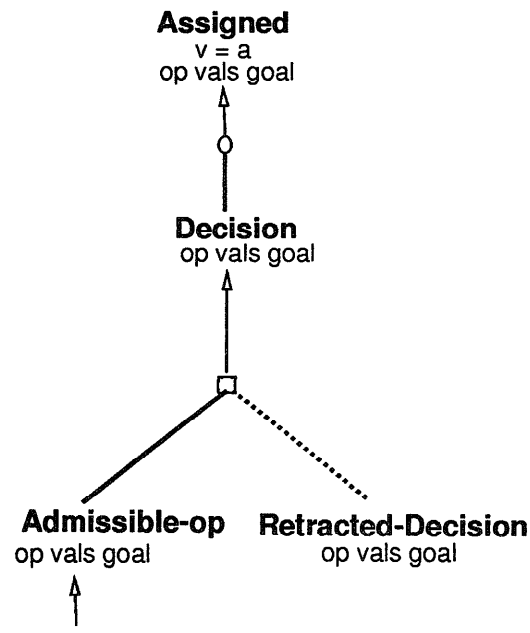

Figure 5: Decision Validity Justification

pendencies generated each time a decision is made in a plan. In this graph, each node may have one or more *justifications* denoted by a directed line. The arrow points to the node being justified. The open circle connects with solid and dotted lines to supporting nodes. If the solid line supporting nodes are valid (IN in TMS language) and none of the dotted line supporting nodes are (i.e., OUT), then the justification is valid and so is the node being supported. For a node to be valid, it must have at least one valid justification. A node such as **Retracted-Decision** initially has no justification and so cannot be valid. Determining these validities for such a graph is the TMS labeling task [2]. It should be emphasized that most of these nodes may have more than one justification and the job of the TMS is to correctly label the graph, thus propagating change.

When a decision is made, the rationale for choosing the operator instance determines the justification for **Best-Op**. This is done in REDUX by converting into a TMS justification a backward chaining proof of the optimality of choosing the operator instance. Horn clause rules are used that have an **Unless** metapredicate encoding negation as failure. When coverted to TMS justifications, these **Unless** clauses become elements of the OUT-lists and are typically used to encode assumptions as described by Doyle[2].

Similarly, the **Admissible-op** node will have a justification based on a proof of the validity of the admissibility conditions for the operator instance. Assumptions here are also encoded via **Unless** clauses converted to OUT-lists. The **Valid-goal** node for the

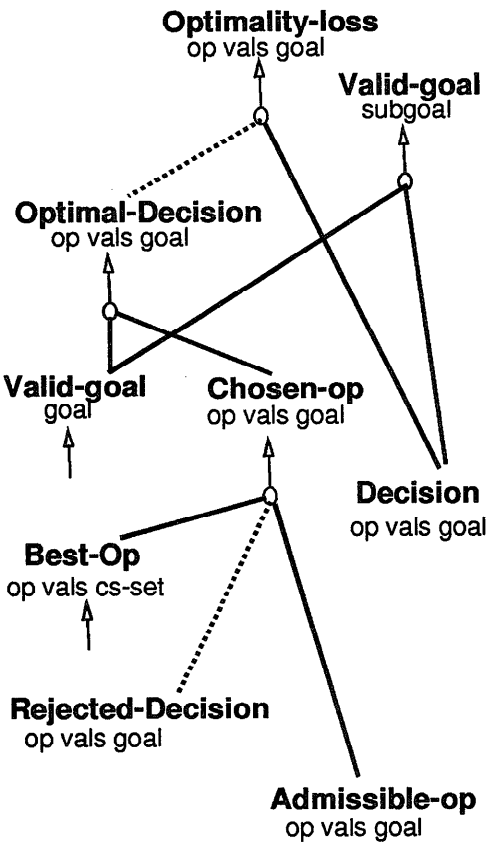

Figure 6: Decision Optimality Justification

parent goal will already have a valid justification; otherwise, making this decision would not be on the task agenda. The validity of each new subgoal depends upon the validity of the parent goal and the decision, as indicated in the justification for the node **Valid-goal**. The node **Optimality-loss** is initially invalid. If it becomes valid, for instance if the decision rationale loses its validity, this signals a possible opportunity.

The **Decision** nodes are always encoded as assumptions because of the **Retracted-Decision** nodes in the OUT-list of their justifications. These retraction nodes may have a *premise*[2] justification provided or removed as a metadecision by the problem solver. This in turn depends upon the validity of **Rejected-Decision** nodes. The justifications for rejection nodes are created in dependency-directed backtracking and correspond to the conditional proof justifications of Doyle[2], and the elective justifications we described in [6]. The CDP model provides these with semantics (tracking pareto optimality) for the first time, as described previously in [8] and in more detail in [9].

Control

We will use the REDUX implementation to describe informally control of CDP solving. There is an agenda with four possible kinds of tasks: satisfying a goal, resolving a constraint violation, responding to an opportunity, and resolving a case in which no operator may be applied to some goal. Control over the agenda is determined by heuristic rules using a *Choose-task* predicate. Partial orderings over operator instances are determined by a *Prefer-operator* predicate. There is an explicit rejection predicate, *Prefer-rejection*, for controlling backtracking. Constraint propagation is controlled by *Propagate-operator* when desired.

By providing a *Prefer-goal* predicate, together with some default heuristics, we find that *Choose-task* and *Prefer-rejection* are rarely used. For example, one default heuristic is: reject first the decision that reduces the lesser preferred goal. Another is: given a choice between the tasks of satisfying two goals, choose the the preferred goal first. (This turns out to implement the variable choice technique in constraint satisfaction.) And in REDUX, subgoals inherit preference from supergoals. All of these *preferences* are encoded as Horn clause rules in REDUX.

REDUX contains other useful default heuristics. For instance, it is generally useful to first resolve a constraint violation before attempting to satisfy any subgoals resulting from the decisions involved in the violation. Since one or more of them may be retracted,[3] it is wise to avoid working to satisfy goals that may become invalid when the decisions are revoked. Another example is that it is generally better to attempt all ways of consistently satisfying a subgoal than to retract immediately the decision creating the subgoal. By incorporating such heuristics, problem representation is simplified in most cases.

Application

The claim of the value of the CDP model illustrated in Figures 2 and 3 is the same as for any other representation and problem solving method: it simplifies formulating and solving a large and useful class of problems: those with the kinds of backtracking and replanning functionality indicated by the travel example. Planning problems with general constraints, multiple objectives, and contingencies should be easier to formulate and solve with a CDP model-based architecture than by using other formalisms or *ad hoc* expert systems. Testing this thesis requires building applications in a CDP-based architecture such as REDUX.

Extended versions of the travel example have been constructed. In [9], we describe two larger applications. One is a telephone network multiplexor configuration application previously done with a commercial expert

[3] One option is to suspend constraint violation resolution and to have an inconsistent plan with known problems.

system. This application has no contingencies but required over 200 Lisp functions for control in addition to many other utilities and shell object definitions. The REDUX implementation provided equivalent control with five (5) operators and fewer than ten rules, assertions, and constraints. We claim that the REDUX implementation was clearer and made it easier to try different search strategies.

As an example, we include here the three (3) main REDUX operators; REFINE, FINISH, and COMPONENT-CHOICE, together with a preference between the operators REFINE and FINISH that encodes component decomposition, using the domain predicates defined by the earlier implementation:

```
REFINE:
instance: operator
variables: (?artifact)
applicable-goal: (Design ?artifact)
  from Required( ?artifact ?bp ?part)
new-assignment: (Part-of ?artifact ?part) )
  from Required( ?artifact ?bp ?part)
new-goal: (Design ?part))
  from Required( ?artifact ?bp ?part)
new-goal: (Select-BP ?artifact ?bp))
  from New-component( ?artifact ?bp)

FINISH:
instance: operator
variables: (?artifact)
applicable-goal: (Design ?artifact)
new-assignment: (terminal-part ?artifact))

prefer-operator( REFINE (?artifact)
                 FINISH (?artifact))

COMPONENT-CHOICE:
instance: operator
variables: (?artifact ?bp ?part)
applicable-goal: (Select-BP ?artifact ?bp)
   from Required-Choice( ?artifact ?bp ?part)
new-assignment: (Part-of ?artifact ?part))
new-goal: (Design ?part))
```

The component decomposition computation starts with the assertion of a goal with the consequent **Design** < *component* >. The *applicable-goal* attribute of the operators REFINE and FINISH will both unify with such a goal. (The *?* denotes a unification variable in the language syntax, distinct from a plan variable.) Because of the preference, REFINE will always be tried first. However, there is a condition on applicability, indicated by the *from* keyword. That the component is decomposable is indicated by proof or assertion of the clause using the **Required** predicate. If this is not the case, then REFINE is not applicable, and FINISH will be used, constructing a leaf node on the decomposition.

If REFINE is used, it uses the **Part-of** predicate to assign the subcomponent (bound to *?part*) and creates two new goals. One is always a recursive generation of the goal **Design** with a subcomponent subcomponent. However, another domain predicate, **New-component** indicates a choice between possible subcomponents. This subgoal, **Select-BP**, is reduced by the COMPONENT-CHOICE operator. There will be several instances of this operator generated by the *from* condition on the *applicable-goal* clause: different choices will bind to the variable *?part*. Not shown are particular preferences among choices that are expressed using it Prefer-operator. COMPONENT-CHOICE recurses to the **Design** goal. Also not shown is the preference (using **Prefer-goal**) for the **Select-BP** goal over the **Design** goal and other preferences expressing such domain control strategies.

The second application was professor and course assignment, emphasizing revision as contingencies occurred during an academic year. The inadequacy of integer programming was described in [1]. The corresponding expert system, based on a research shell with a TMS, had about 60 control rules and was extremely difficult to build and manage.[4] The corresponding REDUX implementation had five (5) operators and four (4) control preferences. Again, the control strategy was clear and easy to experiment with the REDUX implementation. We also found that explanations saying why some operator instance was chosen to reduce a goal from a conflict set of such instances facilitated debugging.

The admissibility conditions of operators were more useful in this application. In a simplified syntax, assumptions are indicated with the keyword *contingency*. Thus, an operator that assigned a teacher to a course in semester would have the condition: `contingency: Unavailable-teacher(?teacher ?sem)`. This would translate into a clause in the OUT-list of the justification supporting an **Admissible-op** node.

Finally, the use of a TMS deserves mention. While it is not necessary for implementing the CDP model, a TMS does provide a convenient and efficient utility for a CDP architecture. We have not done an efficiency analysis, but as long as the decisions are not too globally interdependent, TMS justification caching and label updating seems to take less computation than the backward chaining proofs with the shell we used. And the CDP approach described above avoids the usual burdens imposed on developers using a TMS. Unlike the standard problem solver/TMS model, these dependencies are generated automatically and have semantics. The user need not think about them, nor about the TMS at all; only about the CDP model of problem solving.

For none of the three applications would the use of a classical planner have been appropriate. Nor was

[4] Many more utility rules, assertions, and frames were required for the application, but were not counted because similar constructs were needed for the REDUX application as well. However, these utilities had nothing to do with revision and control.

integer programming or simple constraint satisfaction a good alternative. Expert systems approaches were difficult and cumbersome. We conclude that the CDP model, and the REDUX architecture, facilitates the construction of at least modest CDPs and that there is suggestive evidence to warrant further development and study of larger problems. The CDP model also provides a foundation for the comparison of specialized formalisms, such as HNP and integer programming.

References

[1] Dhar V. and Raganathan N., "An Experiment in Integer Programming," *Communications of the ACM*, March 1990. Also MCC TR ACT-AI-022-89 Revised.

[2] Doyle J., "A Truth Maintenance System," *Artificial Intelligence*, **12**, No. 3, pp. 231-272, 1979.

[3] Feldman, Allan M., *Welfare Economics and Social Choice Theory*, Kluwer, Boston, 1980.

[4] Kambhampati, S., "Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach," University of Maryland, CS-TR-2334, October, 1989.

[5] Morris, P. H., Nado, R. A., "Representing Actions with an Assumption-Based Truth Maintenance System," *Proc. Fifth National Conference on Artificial Intelligence*, AAAI, pp. 13-17, 1986.

[6] Petrie, C., "Revised Dependency-Directed Backtracking for Default Reasoning," *Proc. AAAI-87*, pp. 167-172, 1987. Also MCC TR AI-002-87.

[7] Petrie, C., "Reason Maintenance in Expert Systems," Kuenstliche Intelligence, June, 1989. Also, MCC TR ACA-AI-021-89.

[8] Petrie, C., "Context Maintenance," Proc. AAAI-91, pp. 288-295, July, 1991. Also MCC TR ACT-RA-364-90.

[9] Petrie, C., "Planning and Replanning with Reason Maintenance," U. Texas at Austin, CS Dept. Ph.D. dissertation. Also MCC TR EID-385-91.

[10] Sacerdoti, E., "Planning in a Hierarchy of Abstraction Spaces," IJCAI-73, Palo Alto, CA, 1973 [ABSTRIPS]

[11] Sacerdoti, E., "The Non-linear Nature of Plans," IJCAI-75, Tbilisi, USSSR, 1975. [NOAH]

[12] Tate, A., "Project Planning Using a Hierarchical Non-linear Planner," Dept. of AI Report 25, Edinburgh University, 1977. [NONLIN]